CS406 – Parallel Computing FUTOSHIKI HW2

Oguzhan Ilter
19584
February 2020

1 Introduction

The aim for this assignment is to design and implement an algorithm that can solve efficiently (in terms of run-time and memory consumption) any given solvable Futoshiki Puzzle. In the implementation, there should be parallelization with OpenMP library. SIMD operations and prefetching can be implemented for the sake of providing spatial/temporal locality.

In Futoshiki puzzles, the objective of the game is to discover the digits hidden inside the board's cells; each cell is filled with a digit between 1 and the board's size. On each row and column each digit appears exactly once. Some digits may be given at the start. Inequality constraints are initially specified between some of the squares, such that one must be higher or lower than its neighbor. These constraints must be satisfied in order to complete the puzzle.

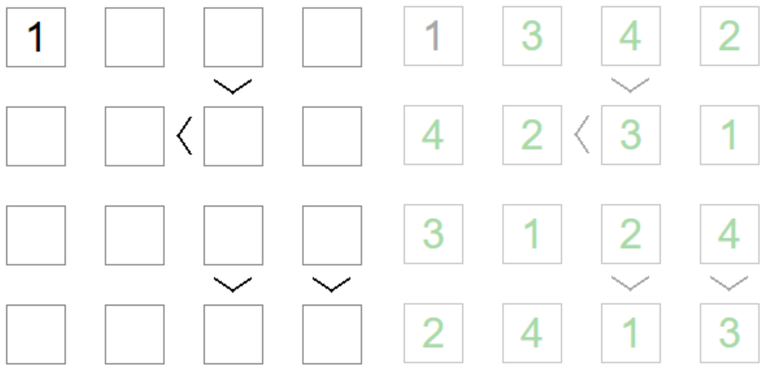


Figure 1: Example of Futoshiki Puzzle and its answer

In the following sections, the implementation steps along with the problems and their solutions are going to be discussed. In addition, execution time of the algorithms will be provided as a graph.

2 Implementation

The implementation is a combination of logical rules. Values of each node restrict the possible number domain of other nodes according to rules that are explained in the previous section. However, in any case, it is a recursive algorithm and relies on the fact, values that are previously assigned to other grids are correct. At every iteration (recursion call), possible values domain of related grids (*biggerThan*, *smallerThan*, *dependedGrids*) are updated according to current (*focus*) grid value. If a possible number domain set of a node becomes empty due to the value of *focus* grid, the recursion *returns false* so that the task finishes its execution and frees the thread. The three main part of the algorithm are *createGridMap*, *parallelRecursiveSolver* and *matrixCallback*.

In this homework, I re-designed the sequential algorithm that is used in hw1, although the logic is still same. In the new algorithm, *column* and *row* arrays have been merged into one *dependedGrids* vector. In addition, all dependency vectors have been turned into integer arrays instead of pointers. The main G*ridMap* structure has been transformed to an array from a matrix. The most significant change is in the representation of possible number domain of a grid. In the previous implementation, the domain was represented by a vector with size of Futoshiki puzzle plus 1. The first index was representing the remaining number of possible numbers of a grid and the remaining indexes were used to represent to which numbers are available to be assigned.

0	1	2	3	4	5
3	0	1	1	0	1

Figure 2: Example of old representation. The grid has 3 possible values: 2, 3 and 5.

In the new implementation, the possible number domain is represented by a short integer. The bit values represent which numbers can be assigned and number of possible numbers is the hop count of the short integer. So that *arrangedomain* function has been reduced to logical bit operations from loops with if statements.

The old algorithm had one main data structure which is called *GridMap* which includes every relationship between grids (*biggerThan*, *smallerThan*, *column*, *row*) along with the grids' attributes (*domain, value, id*). Every change has been carried on this data structure but for parallel cases, it is not possible. Therefore, I created small sized state maps (*miniGridMap*) which contains information about the current values of grids and their domains along with the current empty grids. This structure can be copied to every task without huge cost.

The new solver does not contain any *return* to terminate the process, since in openMP it is not allowed. In addition, a copy mechanism of states has been added via *firstPrivate*. The pseudocode of my recursive puzzle solver is as follows:

```
Algorithm General Structure of Recursive Parallel Solver
Require (miniGridMap)
----------------------------------------------------------
1:  IF emptyGridCount == 0 then
2:      done = TRUE
3:      finalStateofPuzzle = GridMapInUse
4:  ELSE IF(!done) then
5:      →Select focus_grid
6:      → Try to reduce domain
7:      FOR ALL possible values of focus_grid
8:          OMP TASK → firstPrivate(GridMapInUse)
9:          focus_grid.Value ← a value from domain set
10:
11:         IF no empty domain then
12:             solver()                >> recursive call
13:
14:         ELSE→task ends
----------------------------------------------------------
```

Code 1: Pseudocode of puzzle solver

You can check for the graph of my full algorithm in the appendix part.

The following run-time table shows the execution time of each input with different number of threads in seconds. The values are not deterministic since the schedular picks tasks to execute; therefore, the values are the average of 10 runs. I will explain further about the non-deterministic situation.

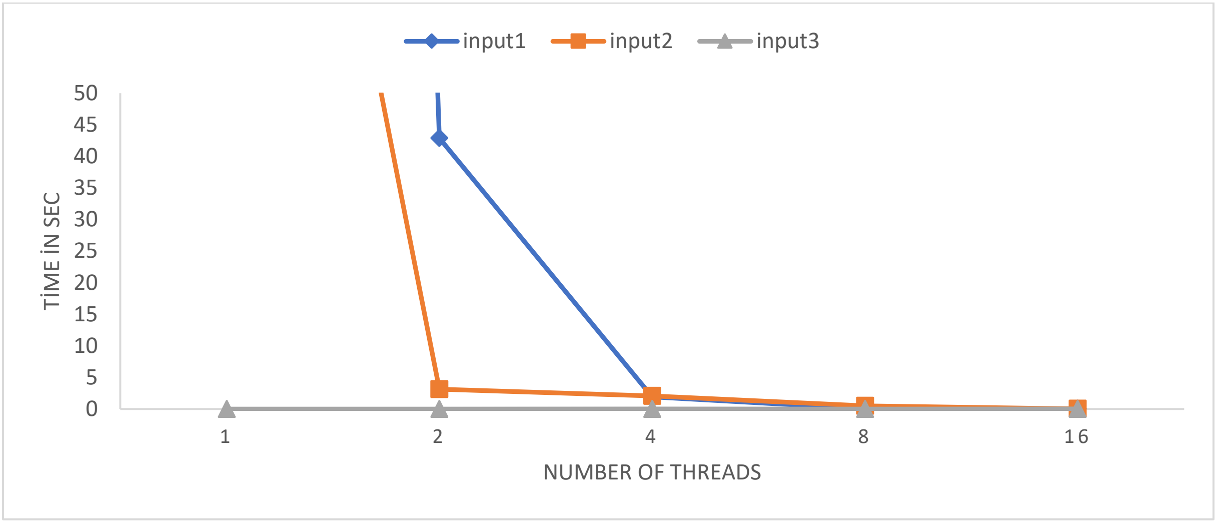
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
Input1	---	42.87102	1.924468	0.033386	0.020709
Input2	174.4273	3.136921	2.057428	0.4728	0.014925
Input3	0.0086774	0.0026405	0.0022639	0.00450262	0.009275

Table 1: Cold Cache, average timing of 10 runs (sec)

The full tables can be found in appendix.

The average values lack of expressing the fluctuation at the timing. The timing difference is acceptable when the number of threads is more than 4. However, when 4 or less threads are used, the two consecutive runs were 0.00141077 sec and 219.065 sec. This huge difference happens once at every 6-7 runs. I will discuss this issue further in comments section.

The graph of the table is as follows.



Graph 1: *Graph of table 1*

The calculated speedup and thread efficiency values are given in the following tables.

Num of Treads	**1**	**2**	**4**	**8**	**16**
Timing (sec)	1000	42.871	1.92447	0.03339	0.02071
Speedup	0.272	6.344615	141.3376	8146.152	13133.75
Efficiency	0.272	3.172308	35.3344	1018.269	820.8595

Table 2: *Timing, speedup, and efficiency table of input 1*

Num of Treads	**1**	**2**	**4**	**8**	**16**
Timing (sec)	174.427	3.13692	2.05743	0.4728	0.01493
Speedup	1.828845	101.6921	155.0478	674.7039	21366.38
Efficiency	1.828845	50.84605	38.76195	84.33799	1335.399

Table 3: *Timing, speedup, and efficiency table of input 2*

Num of Treads	**1**	**2**	**4**	**8**	**16**
Timing (sec)	0.008677	0.002641	0.002264	0.004503	0.00927
Speedup	10372.25	34084.45	39754.41	19988.45	9708.738
Efficiency	10372.25	17042.23	9938.602	2498.556	606.7961

Table 4: *Timing, speedup, and efficiency table of input 3*

In addition to that I want to examine what processes take most of the time. By using *perf record,* I saw that %30-35 of the process time is allocated by malloc operation. Therefore, I wanted to test the algorithm in warm cache cases and see the difference.

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
Input1	---	0.0292229	0.000144974	0.000160609	0.000203738
Input2	3.48635	0.000174008	0.000199972	0.000215174	0.000255085
Input3	0.000364711	0.000203326	0.0002299	0.000245422	0.000284425

Table 5: warm cache runtime average of 50 run

Warm cache case is much faster than cold cache because it does not loose time for memory access.

3 Tricks Done for Efficiency

- **Decreasing Private Data Size**
  In the previous sequential algorithm, I used one data structure to save and track every change during the process. Since this algorithm is parallel, private data structure was needed. Therefore, I created *GridState* class which is much smaller than previous one since it does not contain the relationship between nodes. Thanks to this trick, I have got 10x speedup.

- **Change in Representation of Domain**
  Since I changed the domain representation from vector to integer, in the *arrangeDomain* function, the domains are changed only by bitwise logic operations which very efficient and fast. The total execution time of *arrangeDomain* function is less than 1 microsecond. In addition to that *focus* selection procedure become faster which is also less than 1 microsecond

- **Reducing Branching**
  After selecting *focus grid*, it checks its *dependedGrids'* domains and compare with its own domain. According to this comparison it decreases the search space. This increased the runtime around %10.

4 Compile and Run

I compiled the codes via *gcc 8.2.0* with commend on terminal

```
g++ -std=c++11 -fopenmp xxxx.cpp -g -O3.
```

In addition, I tested their cache miss ratios:

```
perf stat -B -e cache-references,cache-misses,cycles,instructions,L1-dcache-loads,L1-dcache-misses,LLC-loads,LLC-load-misses ./a.out inputs/inputx.txt
```

5 Comments

The algorithm works non-deterministic because after the tasks have been created, scheduler picks a task according to its criteria. This selection leads algorithm to conduct the search in a different branch at every run. I tried giving to tasks priority, but it did not help. One reason of this situation is that the puzzle has multiple correct results. This means that the algorithm may encounter dead end more frequently than the precious homework. However sometimes, a branch which is perfect path to solution can be selected so that the process time decreases significantly. This fluctuation does not happen when the number of threads is more than four threads. Because even some threads go into deep dead-end branches, others continue to search.

Another problem about the algorithm is that it creates more task than it processes. After a result has been found, remaining tasks should be processed. By using if statements, the process time of remaining tasks have been decreased significantly but still the approximately 10% of the process time is remaining processes. I could not find a way to bypass those tasks and finish the process.

To summarize, I must implement some limitations to pick correct task in order to prevent huge fluctuation in run-time but note that there is no fluctuation when I use 8 or 16 threads. In addition, I must come up with a good solution for the remaining tasks. Because this problem causes 10% increase in runtime. My results with 16 threads are as follows: input1→0.02070; input2→0.014925; input→0.009275.

6 Appendix

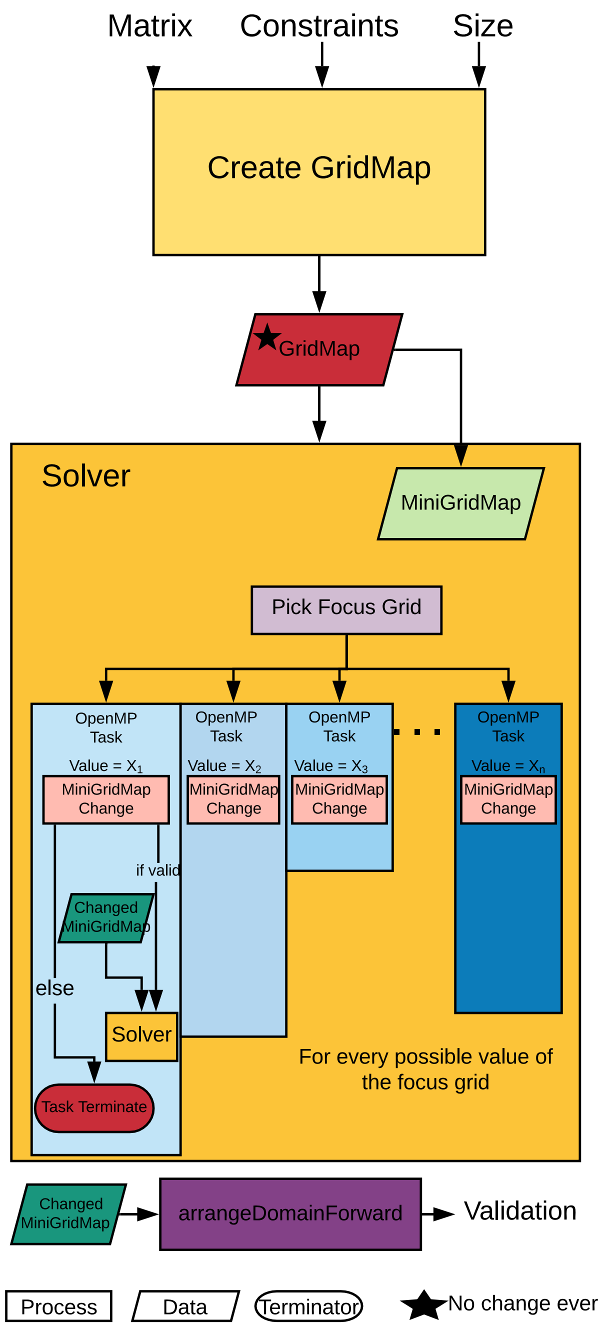


Figure: Process flow and functions

INPUT1	1 thread	2 thread	4 thread	8 thread	16 thread
Run 1		0.011129	0.012689	0.008346	0.010711
Run 2		0.466033	1.22455	0.014616	0.012969
Run 3		0.001411	1.12525	0.019258	0.014832
Run 4		180.752	0.002579	0.004734	0.012459
Run 5		1.09736	7.68257	0.114024	0.011736
Run 6		16.156	8.1237	0.085342	0.011736
Run 7		0.12759	0.773713	0.004988	0.050173
Run 8		219.065	0.013643	0.01126	0.017959
Run 9		1.24706	0.283149	0.063039	0.054122
Run 10		9.78666	0.002841	0.008249	0.010396
Average	500	42.87102	1.924468	0.033386	0.020709
INPUT2	1 thread	2 thread	4 thread	8 thread	16 thread
Run 1	174.48	3.30789	0.002181	0.037615	0.014865
Run 2	173.499	0.179146	0.002514	4.5607	0.014372
Run 3	175.672	3.83978	0.002717	0.005576	0.015195
Run 4	174.792	2.68806	3.94302	0.050628	0.01606
Run 5	173.975	0.001409	8.05767	0.037771	0.014837
Run 6	173.3	8.53228	0.002131	0.00775	0.014893
Run 7	175.6002	0.00842	0.004568	0.005483	0.014837
Run 8	174.892	4.05826	0.002132	0.005756	0.014745
Run 9	173.563	4.37698	4.61433	0.008358	0.014302
Run 10	174.5	4.37698	3.94302	0.008362	0.01515
Average	174.4273	3.136921	2.057428	0.4728	0.014925
INPUT3	1 thread	2 thread	4 thread	8 thread	16 thread
Run 1	0.0085088	0.0014504	0.0027628	0.00411604	0.009437
Run 2	0.0086063	0.0014351	0.0021395	0.00474561	0.009234
Run 3	0.008594	0.0022089	0.002105	0.00412991	0.009598
Run 4	0.0085031	0.0013602	0.0024505	0.00442769	0.009598
Run 5	0.0088397	0.001618	0.0022707	0.00439498	0.009279
Run 6	0.0087848	0.0011627	0.002304	0.00435557	0.009101
Run 7	0.0086056	0.0013854	0.002234	0.00435557	0.008835
Run 8	0.0084942	0.0015009	0.0020003	0.00488012	0.009101
Run 9	0.009032	0.0127148	0.0019851	0.00484163	0.009191
Run 10	0.0088054	0.0015683	0.0023871	0.00477906	0.009375
Average	0.0086774	0.0026405	0.0022639	0.00450262	0.009275

Table: Timing measures